Capítulo 14
Álgebra Lineal

La imaginación es más importante que el conocimiento.

A. Einstein

Objetivos

El objetivo de este capítulo es presentar la biblioteca `numpy`, como herramienta para realizar operaciones de álgebra lineal.

14.1 Introducción

Existen diversos lenguajes y programas que permiten realizar cómputo matemático. Debido a su gran popularidad, en Python se ha desarrollado una serie de herramientas que permiten realizar operaciones con matrices. El objetivo de este capítulo es presentar la biblioteca que permite realizar este tipo de operaciones en Python llamada `pylab`. De hecho `pylab` no es sólamente una biblioteca sino un conjunto de bibliotecas que permiten realizar cómputo matemático.

En este capítulo se presentan ejemplos de aplicaciones con la biblioteca `numpy` que pertenece a `pylab` haciendo uso de algunas otras como `matplotlib` ya utilizadas en este libro.

Para empezar a usar `pylab` es necesario instalarlo siguiendo un proceso similar a la instalación de `numpy` descrita en el Apéndice A.

14.2 Datos de Tipo `array`

Muchas de las funciones de la biblioteca `numpy` regresan el tipo de dato `array`, por lo que en la página oficial (http://scipy.org/) la recomendación es que se realicen los cálculos utilizando este tipo de datos. Mostramos con ejemplos el uso de algunas de las funciones, adicionales a las que se presentaron en capítulos anteriores.

Ejemplo 14.1

Arreglo del 0 al 9 usando `numpy`

La forma de crear los datos rápida y fácilmente utilizando la biblioteca `numpy` es:

```
#Se importa la Biblioteca
import numpy as np
# Creamos una variable que contenga los valores del 0 al 9
a = np.arange(10)
# Desplegamos el valor de la variable
print(a)
```

El resultado es:

```
[0 1 2 3 4 5 6 7 8 9]
```

Teniendo este tipo de dato podemos llegar a preguntarnos qué podemos hacer con él. Lo primero que se quiere saber es la forma y el tamaño del mismo. En el caso de vectores el tamaño es equivalente a la longitud del mismo.

Ejemplo 14.2

Conociendo la forma (`shape`) y tamaño (`size`) de un arreglo

Si una vez creado un arreglo deseamos conocer la forma (`shape`) y el tamaño (`size`) del mismo, se puede hacer de la siguiente forma:

```
>>> # Crear un arreglo multidimensional 2x2 con tipo de dato int32
>>> A = np.array([[1, 2], [2, 4]] , np.int32)
>>> print (A)
 [[1 2]
 [2 4]]
```

```
>>> print(A.shape)
(2, 2)
>>> # Para conocer el tamaño del arreglo
>>> print(A.size)
4
```

14.2.1 Operaciones para crear arreglos

La biblioteca `numpy` cuenta con una diversidad de funciones que pueden usarse para construir arreglos (arrays) con Python. Brevemente explicaremos algunas de ellas.

Ejemplo 14.3

Cambiando la forma de un arreglo utilizando reshape

Para crear una matriz de tamaño 2×5 (o de cualquier tamaño) usando una matriz o vector creado con anterioridad se utiliza la función `reshape`

```
# Se importa la Biblioteca numpy
import numpy as np
# Creamos una variable que contenga los valores del 0 al 9
a = np.arange(10)
# Desplegamos el valor de la variable
print(a)
```

Con lo que obtenemos:

```
[0 1 2 3 4 5 6 7 8 9]
```

y si ahora deseamos convertir este vector en una matriz de 2×5, que tiene un total de 10 elementos igual que el vector original, usamos `reshape` de la siguiente manera:

```
>>> print(a.reshape( 2, 5))
```

para obtener:

```
[ [0 1 2 3 4]
[5 6 7 8 9] ]
```

La función `reshape` solamente funciona si el número de elementos del arreglo previo y el del arreglo final son iguales. Si usamos, por ejemplo, `a.reshape( 3, 3)` obtenemos un error, ya que el arreglo resultante solamente tiene 9 elementos:

```
>>> a.reshape(3,3)
Traceback (most recent call last):
   File "<pyshell#18>", line 1, in <module>
      a.reshape(3,3)
ValueError: total size of new array must be unchanged
```

Existe la posibilidad de crear vectores fila con numeración ascendente o descendente. Utilizando el comando `arange` se pueden generar vectores de tipo incrementales y decrementales. La sintaxis es la siguiente:

```
arange(valor_inicial, valor_final, incremento)
```

El arreglo que se obtiene empieza en `valor_inicial`, termina antes de `valor_final`, es decir, NO se incluye este valor en el arreglo, y los valores intermedios están separados por el valor del incremento. Si tenemos:

```
vector = arange( -2, 10, 3)
```

entonces obtenemos que vector está dado por:

```
[ -2, 1, 4, 7 ]
```

donde vemos que no se incluye el valor final que es 10, mientras que los valores están separados por el incremento.

La función `arange` también se puede usar si el incremento es negativo, es decir, hablamos de un decremento. Como ejemplo para `arange( 5, -2, -1)` obtenemos:

```
[ 5, 4, 3, 2, 1, 0 ]
```

y no se incluye el -1 ya que es el valor final.

Cabe mencionar nuevamente que para poder usar las instrucciones de éste capítulo como la instrucción `arange` se necesita importar la bibioteca `numpy` al comienzo del script.

Ejemplo 14.4

Arreglos con numeración ascendente o descendente usando `numpy`

Se desea crear un arreglo que inicie en 0, termine antes del 10 y que tenga un incremento de 0.2. Además, se desea obtener otro vector que inicie en 20, termine antes de 10 con decremento de -1. Para hacerlo usamos:

```
import numpy as np
a = np.arange(0, 10, 0.2)
b = np.arange(20, 10, -1)
print(a)
print(b)
```

El resultado se obtiene con `print`. Para `a` es:

```
[ 0. 0.2 0.4 0.6 0.8 1. 1.2 1.4 1.6 1.8 2. 2.2 2.4 2.6 2.8
 3. 3.2 3.4 3.6 3.8 4. 4.2 4.4 4.6 4.8 5. 5.2 5.4 5.6 5.8
 6. 6.2 6.4 6.6 6.8 7. 7.2 7.4 7.6 7.8 8. 8.2 8.4 8.6 8.8
 9. 9.2 9.4 9.6 9.8]
```

y para `b`:

```
[20 19 18 17 16 15 14 13 12 11]
```

Existen también funciones que permiten crear matrices definidas y conocidas de álgebra lineal como la matriz identidad o matrices con ceros. La tabla 14.1 muestra de manera condensada estas funciones.

Tabla 14.1 Operaciones para crear Matrices

Comando	Descripción
`eye(n)`	Crea una matriz de tamaño `n` con unos en la diagonal principal y ceros en las otras posiciones.
`ones(m, n)`	Crea una matriz de tamaño `m`×`n` llena de unos.
`zeros(m, n)`	Crea una matriz de tamaño `m`×`n` llena de ceros.
`empty(m, n)`	Crea una matriz de tamaño `m`×`n` vacía.
`rand(m, n)`	Crea una matriz de tamaño `m`×`n` con números aleatorios.
`max(a)`	Obtiene el máximo entre los valores de un arreglo.
`min(a)`	Obtiene el mínimo entre los valores de un arreglo.
`diag(a)`	Genera un arreglo con los elementos de la diagonal de `a`.
`linspace(a, b, c)`	Crea un arreglo de muestras espaciadas a intervalos regulares dados por `c` entre `a` y `b`.

Ejemplo 14.5

Creación de matrices usando `numpy`

Para utilizar las funciones para crear matrices basta seguir los ejemplos siguientes:

Para crear las matrices de 3×3 identidad, 3×5 de ceros y 4×6 de unos usamos:

```
import numpy as np
I = np.eye(3)
b = np.zeros(3, 5)
o = np.ones(4, 6)
 print(I)
 print(b)
 print(o)
```

Con lo que obtenemos, para la matriz identidad de dimensión 3×3:

```
[[ 1. 0. 0.]
[ 0. 1. 0.]
[ 0. 0. 1.]]
```

Para la matriz de 0's de dimensión 3×5, obtenemos:

```
[[ 0. 0. 0. 0. 0.]
 [ 0. 0. 0. 0. 0.]
 [ 0. 0. 0. 0. 0.]]
```

y para la matriz de 1's de dimensión 4×6 tenemos:

```
[[ 1. 1. 1. 1. 1. 1.]
 [ 1. 1. 1. 1. 1. 1.]
 [ 1. 1. 1. 1. 1. 1.]
 [ 1. 1. 1. 1. 1. 1.]]
```

Python como lenguaje de programación inicia sus arreglos en el índice 0. Al respecto de los índices, Python nos permite con ellos realizar búsquedas de elementos en los arreglos de forma muy sencilla. El ejemplo 14.6 muestra como realizar la búsqueda de datos usando los índices de las matrices.

Ejemplo 14.6

Manejo de los datos dentro de matrices usando índices

Se desea obtener renglones o filas y columnas de una matriz. Para esto vamos a generar una matriz de 6×7 con múmeros aleatorios.

```
import numpy as np
A = np.random.rand(6, 7)
print("Mostramos la primera y segunda columnas", A[:, 0:2])
A = np.arange(15).reshape(3, 5)
print("Mostramos la ultima fila", A[-1]) [10 11 12 13 14]
[10 11 12 13 14]
print("Mostramos las ultimas 3 columnas", A[:, -3:])
 [ 7 8 9]
 [12 13 14]]
print(''Tomando una matriz menor dentro de A'', A[0:3, 3:6])
[ 8 9]
 [13 14]]
```

14.3 Tipo `matrix`

Menos utilizado y usado sólo para compatibilidad con el álgebra lineal, el tipo `matrix` es similar en muchos aspectos a la forma de trabajar en otros lenguajes, las limitantes más importantes es que para crear matrices multidimensionales, éstas deben de crearse como tipo `array` y después hacer un `cast` al tipo `matrix`. Para ver el uso de este tipo de datos usaremos un ejemplo:

Ejemplo 14.7

Creando Matrices usando el tipo de dato matrix en `numpy`

Supongamos que se tiene un problema que se resuelve con la multiplicación de una matriz por un vector, utlizando tipo de datos `matrix`. Sea la matriz `A` dada por:

$$A = \begin{bmatrix} 4 & -3 & 11 & 1 \\ 5 & 9 & 7 & 2 \\ 2 & 3 & 4 & 1 \\ 5 & 3 & -5 & -9 \end{bmatrix} \tag{14.1}$$

y el vector columna B:

$$B = \begin{bmatrix} 4 \\ 9 \\ 1 \\ 3 \end{bmatrix} \tag{14.2}$$

Para hacerlo en Python utilizando `numpy` hay que realizar lo siguiente:

```
import numpy as np
A = np.matrix("4 -3 11 1; 5 9 7 2; 2 3 4 1; 5 3 -5 -9")
B = np.matrix("4; 9; 1; 3")
print( " Realizamos el producto matricial A*B",  A*B)
[ [ 3]
 [114]
 [ 42]
 [ 15] ]
```

Vemos que el resultado es un vector columna.

14.4 Operaciones con matrices

En la Tabla 14.2 tenemos una lista de las operaciones elementales utilizadas para manejar matrices. A lo largo del capítulo se hará uso de éstas y otras operaciones en los diversos ejemplos.

Tabla 14.2 Operaciones con matrices usando Python

Operación	Operandos	Comando
Suma	$A_{\{m\times n\}}$, $B_{\{m\times n\}}$	`A + B`
Producto	$A_{\{m\times n\}}$, $B_{\{n\times p\}}$	`np.dot(A, B)`
Inversa	$A_{\{m\times n\}}$	`A.inv`
Producto por un escalar	$a \in R$, $A_{\{m\times n\}}$	`a*A`
Determinante	$A_{\{n\times n\}}$	`np.linalg.det(A)`
Transposicion	$A_{\{n\times n\}}$	`A.transpose` ó `A.T`
Rango	$A_{\{m\times n\}}$	`np.rank(A)`
Traza	$A_{\{m\times n\}}$	`np.trace(A)`
Norma	$A_{\{m\times n\}}$	`np.norm(A)`

14.5 Resolviendo Sistemas de Ecuaciones Lineales

En las matemáticas y en específico en álgebra lineal, existen problemas que se caracterizan porque diversas variables se encuentran relacionadas unas con otras a través de conjuntos de ecuaciones.

Cuando cada ecuación dentro de este conjunto de ecuaciones es de primer orden, el sistema es llamado de ecuaciones lineales, o simplemente sistema de ecuaciones.

Los sistemas de ecuaciones lineales tienen una amplia diversidad de aplicaciones en los campos más variados de la ciencia y las matemáticas. Algunos ejemplos son

- Procesamiento digital de señales
- Análisis estructural
- Análisis de circuitos
- Regresión Lineal

Diversos métodos han sido desarrollados para resolver sistemas de ecuaciones:

por sustitución, Regla de Cramer, Gauss-Jordan, matrices inversas así como distintos algoritmos numéricos.

Para comprobar la forma en la que Python y `numpy` trabajan con matrices se realizará el ejemplo a continuación.

Ejemplo 14.8

Resolviendo Sistemas de Ecuaciones

```
1  -3  1.   x.     2
(1   1   1)( y) = ( 5)
-1  9   7   z.     7
```

Sea el siguiente sistema de ecuaciones:

$$\begin{aligned} x - 3y + z &= 2 \\ x + y + z &= 5 \\ -x + 9y + 7z &= 7 \end{aligned} \qquad (14.3)$$

```
Ax = b
A^(-1)Ax = A^(-1)b
Ix = A^(-1)b —>
x = A^(-1)b
```

Realizamos un script de Python con el código necesario para resolver este sistema de ecuaciones,

```
#
# # # # # # # # # #
# Resolviendo un sistema de ecuaciones utilizando
# matriz inversa utilizando la librería numpy.
# # # # # # # # # #
import numpy as np
# primero se declara la matriz de Coeficientes
# como valores de 32 bits
A = np.array([[ 1, -3, 1] , [1, 2, 3], [ -1, 9, 7]] , np.int32)
# seguido de la matriz de resultados
y = np.array([[ 2] , [5], [12] ], np.int32)
# Para calcular la inversa primero verificamos que
# la matriz sea no singular. Obtenemos el Determinante
# y verificamos que este sea distinto de cero
print(“ \n Tenemos el determinante de la matriz \n”)
print(np.linalg.det(A))
if (np.linalg.det(A)) ! = 0.0 :
     # Verificar que el determinante no sea cero
      print(“\n Generamos la matriz Inversa \n”)
      B = np.linalg.inv(A)
```

```
 print(B)
      print("\n Mostramos la multiplicación de y*inv(A) \n")
      print(''\n lo cual resuelve la ecuación. \n'')
      print(np.dot(B , y) )
      print(B.dtype)
else:
      print("\n La matriz de coeficientes es singular. \n")
      # Matriz singular
      print("\n No se puede encontrar una solución.\n")
```

El resultado de ejecutar el script nos permite encontrar el valor de `x`, `y`, `z` es:

```
Tenemos el determinante de la matriz
28.0
Generamos la matriz inversa
[[-0.46428571 1.07142857 -0.39285714]
 [-0.35714286 0.28571429 -0.07142857]
 [ 0.39285714 -0.21428571 0.17857143]]
Mostramos la multiplicacion de y*inv(A)
[[-0.28571429]
 [-0.14285714]
 [ 1.85714286]]
float64
```

Cabe hacer una observación, al realizar la matriz inversa de la matriz de coeficientes el tipo de dato pasa de ser tipo de dato enteros de 32 bits a flotante de 64 bits.

14.6 Regresión lineal

Un ejemplo de una aplicación práctica de los sistemas de ecuaciones lineales es la aproximación lineal por el método de mínimos cuadrados. Como nota histórica cabe mencionar que este elegante método fue inventado por el genio incansable de Carl Friedrich Gauss a la edad de 17 años.

El método de mínimos cuadrados se implementa cuando se desea obtener la relación matemática entre las variables de un problema. Esta relación podría ser lineal, cuadrática o cúbica. El ejemplo 14.9 utiliza la aproximación lineal de datos.

Ejemplo 14.9

Aproximación lineal usando `numpy`

Se tiene un experimento, que mide la resistividad de un material cuando se somete a un voltaje entre 1 y 10 volts y la corriente que lo atraviesa, el experimento arroja los resultados que se ven en la Tabla 14.3.

Tabla14.3 Resultados del experimento.

Voltaje (mV)	Corriente (uA)
1	-7.4
2	-2.3
3	3
4	7.6
5	12.0
6	17.9
7	22.5
8	27.6
9	32.1
10	37.4

La gráfica de la figura 14.1 fue creada en Python también. El código se muestra a continuación:

```
import matplotlib.pyplot as plt
import numpy as np
x = np.arange(1, 11)
y = [-7.4, -2.3 , 3 , 7.6 , 12 , 17.9 , 22.5 , 27.6 , 32.1, 37.4 ]
plt.plot(x, y, 'ro')
```

```
plt.xlabel('Voltaje')
plt.ylabel('Corriente')
plt.title('Resultados Graficados')
```

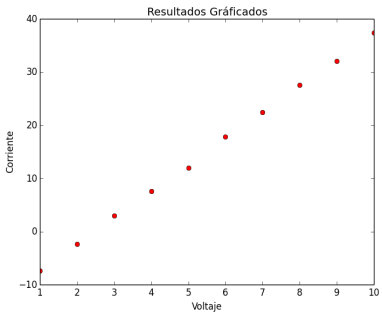

Figura 14.1 Gráfica de los datos.

Se quiere conocer relación de la forma $y = f(x) = ax + b$, si existe y que minimice el error de los siguientes datos. Se desea que la ecuación anterior pase por cada punto de la lista anterior entonces es necesario que se cumpla.

- $-7.4 = a(1) + b$
- $-2.3 = a(2) + b$
- $3 = a(3) + b$
- $7.6 = a(4) + b$
- $12 = a(5) + b$
- $17.9 = a(6) + b$
- $22.5 = a(7) + b$
- $27.6 = a(8) + b$
- $32.1 = a(9) + b$
- $37.4 = a(10) + b$

1 1. -7.2
2 1. -2.3
3 1. .
4 1. a. .
(5 1) (b) = (. .)
6 1
7. 1
8 1
9. 1
10 1. 37.4

Las ecuaciones anteriores definen un sistema de 10 ecuaciones con dos incógnitas, a y b. Para resolver el sistema de ecuaciones lo intentaremos de dos formas distintas.

El sistema puede representarse como el producto de dos matrices

$$A_{(10,2)} * X_{(2,1)} = Y_{(10,1)}.$$

Por un lado se resolverá el sistema de ecuaciones utilizando las características de las operaciones matriciales.

La matriz A es una matriz asimétrica, por lo que los procedimientos de matriz inversa o regla de Cramer no pueden ser utilizados, pero si se premultiplica el sistema

$$A * X = Y \tag{14.4}$$

por la matriz transpuesta de A que es A' se tiene el siguiente resultado:

$$(A' * A) * X = (A' * Y) \tag{14.5}$$

Haciendo $S = A' * A$ y $Z = A' * Y$ se obtiene un nuevo sistema de ecuaciones equivalente al primero, de la forma:

$$S * X = Z \tag{14.6}$$

Con el cual se puede, debido a que S es no singular, encontrar una única solución del sistema. Esto se hace utilizando la matriz inversa S^{-1}.

$$(S^{-1} * S) * X = S^{-1} * Z \tag{14.7}$$

$$I * X = S^{-1} * Z \tag{14.8}$$

Hagamos esto en Python:

```
#####################
#
# Regresión lineal
# Utilizando la librería numpy
```

```
######################

import numpy as np

A = np.array([[1, 1], [2, 1], [3, 1], [4, 1],
 [5, 1], [6, 1], [7, 1], [8, 1], [9, 1], [10, 1]], np.float64)
y = np.array([[-7.4], [-2.3], [3], [7.6], [12.0],
 [17.9], [22.5], [27.6], [32.1], [37.4]], np.float64)
# Presentamos el arreglo
print(" Estas son las matrices A y Y del sistema \n")
print(" A = \n")

print(A)
print("Y = \n")

print(y)
# Presentamos la transpuesta de la matriz

print("\n Esta es la matriz transpuesta de A \n")
print(A.T)
S = np.dot(A.T, A)
print("\n Obtenemos la matriz S \n")
print(S)

Z = np.dot(A.T, y)

print("\n Obtenemos la matriz Z \n")
print(Z)

# Escribimos el determinante
print("\n Tenemos el determinante de la matriz S \n")
print("\n distinto de cero \n")
print(np.linalg.det(S))
```

```
# Escribimos la matriz Inversa
print(‘‘\n Presentamos la matriz Inversa \n’’)
print(np.linalg.inv(S))

print(“\n Mostramos los valores de a y b que resuelven el sistema \n”)

X = np.dot(np.linalg.inv(S), Z)
print(X)
```

El resultado del script anterior se presenta de la siguiente forma:

```
Esta es la matriz A y el vector Y del sistema:
 A =
[[ 1. 1.]
 [ 2. 1.]
 [ 3. 1.]
 [ 4. 1.]
 [ 5. 1.]
 [ 6. 1.]
 [ 7. 1.]
 [ 8. 1.]
 [ 9. 1.]
 [ 10. 1.]]
 Y =
[[ -7.4]
 [ -2.3]
 [ 3.0 ]
 [ 7.6]
 [ 12.0 ]
 [ 17.9]
 [ 22.5]
 [ 27.6]
 [ 32.1]
 [ 37.4]]
 Esta es la matriz transpuesta de A
[[ 1. 2. 3. 4. 5. 6. 7. 8. 9. 10.]
```

```
 [ 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]]
 Obtenemos la matriz S
[[ 385. 55.]
 [ 55. 10.]]
 Obtenemos la matriz Z
[[ 1236. ]
 [ 150.4]]
 Tenemos el determinante de la matriz S, distinto de cero
825.0
 Presentamos la matriz Inversa
[[ 0.01212121 -0.06666667]
 [-0.06666667 0.46666667]]
 Mostramos los valores de a y b que resuelven el sistema
[[ 4.95515152]
 [-12.21333333]]
```

Por lo tanto la función que pasa por los puntos, minimizando el error cuadrático medio es:

$$y(x) = 4.95x - 12.21$$

Ejemplo 14.10

Aproximación lineal usando la función `lstsq` de `numpy`

La otra forma de realizarlos es utilizando la función `lstsq`, que es el método de mínimos cuadrados (least squares). El nombre de la función indica que se realiza una regresión lineal por mínimos cuadrados (linear least squares). Veamos el código:

```
############################### #
# Gráfica de Regresión lineal utilizando la función lstsq
# de la biblioteca numpy
# ##############################
```

```
import numpy as np
import matplotlib.pyplot as plt
A = np.array([[1, 1], [2, 1], [3, 1], [4, 1], [5, 1],
 [6, 1], [7, 1], [8, 1], [9, 1], [10, 1]], np.float64)
y = np.array([[-7.4], [-2.3], [3], [7.6], [12.0],
 [17.9], [22.5], [27.6], [32.1], [37.4]], np.float64)
x = A[:, 0]
print(x)
#Con la información de las matrices hacemos la regresión lineal
# y obtenemos los datos de la recta,
#la pendiente junto con la elevación sobre el eje y
a, b = np.linalg.lstsq(A, y)[0]
print(a)
print(b)
plt.plot(x, y, 'o', label = ' Experimentos', markersize = 15)
plt.plot(x, a*x+b, 'r', label = ' Regresion', markersize = 15)
plt.legend( )
plt.show( )
```

El código anterior al ser ejecutado muestra el siguiente resultado:

```
Resultados de la función lstsq:
[ 4.95515152]
[-12.21333333]
```

Como se observa en la salida, los datos obtenidos por la función `lstsq` son iguales a los obtenidos anteriormente. La Figura 14.2 muestra el resultado.

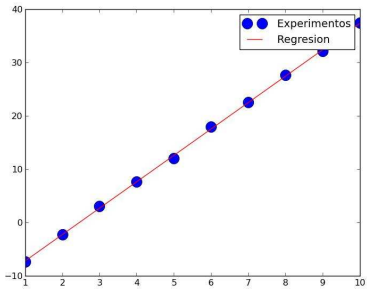

Figura 14.2 Obtención de la función $y(x) = 4.9x - 12.21$.

14.7 Vectores en el espacio de n dimensiones

La notación matemática de sumatoria para referirse a los vectores en el espacio R^n:

$$\mathbf{v} = v_1e_1 + v_2e_2 + v_3e_3 + v_4e_4 + \ldots + v_ne_n$$

es ampliamente utilizada en la literatura. Sin embargo, diversos autores utilizan también la notación de *n-ada ordenada*. Esta representación, característica del álgebra lineal, permite asociar un `array` de dimensión $1 \times n$ en Python a un Vector en R^n.

Como se mencionó anteriormente, se pretende que la clase *array* en `numpy` sea un arreglo n-dimensional de uso generalizado para el cálculo numérico. De esta forma la forma de crear un vector, en coordenadas rectangulares, es similar a los ejemplos previamente expuestos para matrices utilizando la clase `array` .

14.7.1 Longitud de un vector en R^n

Los vectores son entidades matemáticas en el espacio n-dimensional. Existen magnitudes físicas, como la fuerza, que se representan utilizando vectores. La norma de un vector es un número real positivo proporcional a la magnitud que representa.

La clase `numpy` posee la operación `norm` que permite obtener la norma o longitud de un vector.

Ejemplo 14.11

Obteniendo la norma de un vector

Se desea obtener la norma de un vector generado con `arange` que va de 0 a 5. El código es:

```
import numpy as np
v = np.arange(6)
np.linalg.norm(v)
```

El valor de la norma es:

```
7.416198487095663
```

14.7.2 Producto Interno o Producto punto en R^n

El producto punto de dos vectores en `numpy` se puede obtener de varias formas, como con las operaciones: `.dot`, `.vdot` y `.inner`. Estas instrucciones permiten obtener el mismo resultado. Así el código del siguiente ejemplo produce resultados similares para dos vectores en R^6.

Ejemplo 14.12

Obteniendo el producto punto entre dos vectores

Se generan dos vectores con `arange` y se obtiene el producto escalar. El código siguiente produce el resultado deseado:

```
import numpy as np
v = np.arange(6)
u = np.arange(3, 9)
np.dot(u, v)
np.vdot(u, v)
np.inner(u, v)
```

```
print(a,'\t',b,'\t',c)
```

y el resultado que se obtiene es:

```
100     100     100
```

14.7.3 Ángulo entre dos vectores en R^n

EL ángulo θ entre los vectores u y v, ambos vectores R^n, se obtiene a partir de la ecuación:

$$\cos\theta = \frac{u \cdot v}{\| u \| \| v \|} \tag{14.9}$$

Ejemplo 14.13

Obteniendo el ángulo entre dos vectores

Dados dos vectores en R^3: $u = 3i - 2j + 4k$ y $v = i + 2j - 3k$, encuentre el ángulo entre ellos. El código en Python que lo realiza es el siguiente:

```
import numpy as np
u = np.array([3, -2, 4])
v = np.array([1, 2, -3])
teta = np.arccos(np.dot(u, v)/(np.linalg.norm(u)*np.linalg.norm(v)))
teta= np.degrees(teta)
print(teta)
```

El resultado es:

$\theta = -40.18$.

El resultado es un ángulo en grados ya que se convierte antes de la instrucción `print` de radianes a grados.

14.7.4 Producto Cruz o Vectorial en R^3

El producto vectorial o producto cruz de dos vectores u y v, a diferencia del producto interno, el cual produce un escalar, tiene como resultado un vector. Una de las características de dicho vector es que resulta ser perpendicular al plano generado por los vectores u y v.

Si ambos vectores son no nulos, la única forma de que el vector resultante sea nulo es que tanto u como v no formen un plano y que por lo tanto sean colineales.

La biblioteca `numpy` cuenta con una función que permite obtener el producto vectorial de manera sencilla. La función `cross(u, v)` devuelve un `array` con el producto vectorial.

Ejemplo 14.14

Obteniendo producto Cruz entre dos vectores

Usando los vectores del ejercicio anterior tenemos:

```
import numpy as np
u = np.array([3, -2, 4])
v = np.array([1, 2, -3])
C = np.cross(u, v)
print(C)
```

Que nos produce el resultado:

```
[-2 13 8]
```

Otro ejemplo con dos vectores no nulos cuyo resultado es nulo es el siguiente.

Ejemplo 14.15

Obteniendo producto Cruz entre dos vectores

Se desea obtener el producto cruz o vectorial de dos vectores.

```
import numpy as np
u = np.array([1, -9, 14])
v = np.array([-4, 36, -56])
C = np.cross(u, v)
print(C)
```

El resultado es 0 ya que el segundo vector se obtiene del primero multiplicandolo por -4 y por tanto ambos vectores son colineales.

14.8 Eigenvalores y Eigenvectores de Matrices

Se conoce como valores propios o eigenvalores y vectores propios o eigenvectores a los escalares λ y a los vectores v que cumplan para una matriz cuadrada A de tamaño $n \times n$ la siguiente expresión:

$$Av = \lambda v \tag{14.10}$$

El vector v se llama vector propio de A asociado al valor propio λ. `numpy` cuenta con una función llamada `eig` que permite obtener los valores y vectores propios de una matriz cuadrada. La estructura de la función es:

```
L, V = np.linalg.eig(A)
```

Siendo `L` el conjunto de todos los valores propios y `V` el conjunto de los vectores propios de la matriz `A`.

La siguiente sección trata sobre las cadenas de Markov, un proceso estocástico que utliza los eigenvalores de una matriz para obtener las probabilidades de un evento que depende solamente de un evento inmediato anterior.

14.8.1 Cadenas de Markov

Una matriz estocástica es una matriz con dos propiedades, la primera es que sus elementos son no negativos, la otra propiedad es que los valores de cada columna suman 1.

Dentro de las aplicaciones más utilizadas de estos sistemas, que son discretos, existe una en la que se trabaja con cantidades porcentuales que pasan de un estado a otro sin variar en su conjunto. Supongamos el siguiente ejemplo.

Ejemplo 14.16

Uso de matrices para resolver cadenas de Markov

En un sistema bipartidista, en cierto año se calcula que la población de una ciudad de 10000 habitantes está dividida entre el partido A y el B.

En la tabla siguiente se resumen los resultados de un estudio, los cuales pueden interpretarse de la siguiente manera. El primer elemento de la matriz indica que el 51 % de la población que votó por el partido A volverá a votar por ese partido en la siguiente elección, el dato inferior de esa columna indica los votantes que cambiarán de partido en la elección del año siguiente.

La siguiente columna indica en primer lugar cuales votantes cambiarían del partido B al partido A y por último cuales quedarían en el partido B.

Porcentaje de votantes fieles y cambiantes.

	A	B
A	0.51	0.6
B	0.49	0.4

Así, escrita como un sistema de ecuaciones, esta tabla permite predecir para cada año el número de votantes de cada partido, todo esto sin que cambie el total de votantes de la ciudad.

Si se denota por A_{k+1} y B_{k+1} el número de votantes de cada partido para la siguiente elección, entonces tomando en cuenta la tabla anterior se llega al siguiente sistema de ecuaciones:

$$\begin{aligned} A_{k+1} &= 0.51A_k + 0.6B_k \\ B_{k+1} &= 0.49A_k + 0.4B_k \end{aligned} \tag{14.11}$$

que puede escribirse en forma matricial como:

$$X_{k+1} = AX_k$$

donde:

- A es conocida como matriz de transición de estado.
- X_{k+1} y X_k son conocidos como el estado siguiente y actual del sistema, respectivamente.

¿Qué sucede si deseamos conocer el estado del electorado, no en la siguiente elección sino en las dos posteriores? Vemos que $X_{k+2} = AX_{k+1}$. Pero si $X_{k+1} = AX_k$, entonces $X_{k+2} = AX_{k+1} = A^2X_k$.

Como se observa al hacer la operación A^2 se obtiene

$$A^2 = \begin{pmatrix} 0.554 & 0.546 \\ 0.446 & 0.454 \end{pmatrix} \tag{14.12}$$

la cual como se observa es también una matriz estocástica. Esta clase de sistemas se conocen como procesos sin memoria o procesos de Markov, porque el estado futuro del sistema solamente depende del estado actual del mismo y no por los estados pasados.

¿Qué sucede si hacemos el proceso anterior pero hacia atrás? En ese caso tenemos:

$$X_k = AX_{k-1} = A^2X_{k-2} = A^3X_{k-3}... = A^KX_0 \tag{14.13}$$

Ésto nos indica que dado un estado inicial llamado X_0 es posible alcanzar el estado X_k usando la parte final de la ecuación anterior.

De este ejemplo se puede tomar más información que nos ayude a comprender los procesos de Markov.¿Qué sucederá en la quinta o la octava elección? y ¿después de veinte años? Empezaremos a analizar con un escenario donde ambos partidos tienen de inicio el mismo número de electores $X_0 = [5000\ 5000]$ y ahora busquemos el resultado después de 5 años:

```
# ########################
```

```
#
# Resolviendo un proceso de Markov
# Utilizando la libreria numpy.
#
########################
import numpy as np
from numpy import linalg as LA
# Primero se declara la matriz de transición, como números de 32 bits.

A = np.array([[ 0.51, 0.6], [ 0.49, 0.4]] , np.float)
# seguido de la matriz de estado inicial X0
x0 = np.array([[5000] , [5000] ], np.int32)
# Para calcular A^5 se realiza lo siguiente
y = LA.matrix_power(A, 5)
xk = np.dot(y, x0)
print("El numero de votantes por partido despues de cinco años es:")
print(xk.round())
```

El resultado que arroja Python es:

```
print(''El numero de votantes por partido después de cinco años es:'')
[[ 5505.0]
 [ 4495.]]
```

Y después de 20 años los valores son:

```
El numero de votantes por partido después de veinte años es:
[[ 5505.]
 [ 4495.]]
```

Vemos que los resultados ya no varían en el tiempo. En realidad sí hay algunas mínimas variaciones pero se deja al lector verificar esto último (Puede hacerlo si retira el método round( ) del listado anterior).

Ahora empecemos con otro escenario. El Partido A tiene 10000 electores y el B ninguno y vayamos hasta los 20 años. En este caso el vector X_0 es: $X_0 = [10000\ 0]$. Corremos el algoritmo con este valor de X_0 para obtener:

```
El número de votantes por partido después de veinte años es:
[[ 5505.]
 [ 4495.]]
```

lo cual nos sugiere que para cualquier X_0 tal que la suma de sus elementos sean 10000 el resultado es el mismo. De hecho será el mismo no importando el número que se elija en el estado inicial. Esto sucede porque los procesos de Markov, que dependen del tiempo, poseen un vector de estado estable q.

La teoría del álgebra lineal, particularmente el teorema de Frobenius, nos enseña que al menos un eigenvalor de una Matriz de Markov es 1. El eigenvalor 1 esta asociado precisamente al vector en estado estable de la matriz P. Esto permite obtener el valor $X_k = A^K X_0$ cuando $k \to \infty$ sin tener que evaluar la potencia de la matriz de transición.

```
import numpy as np
A = np.array([[ 0.51, 0.6], [ 0.49, 0.4]] , np.float)
D, V = np.linalg.eig(A)
# Con eig se obtienen los eigenvectores y los eigenvalores
# de la Matriz que es el argumento
# Imprimimos V para conocer los dos eigenvectores
print(“ Eigenvalores de la matriz de transición ”)
print(D)
# El eigenvector asociado a lambda es
print(“El eigenvector asociado a lambda = 1 es: ”)
print(V[:, 0])
print(“normalizando el vector y multiplicando por el número”)
print(“de votantes se obtiene el dato que se buscaba.”)
print(np.round(10000*V[:, 0]/sum(V[:, 0])))
```

Lo cual nos da el mismo valor de X_k obtenido anteriormente.

14.9 Instrucciones de Python del Capítulo 14

Instrucción	Descripción
`array`()	Crear tipos de dato `array`, vectores y matrices.
`arange`()	Crear arreglos.
`reshape`()	Modifica el tamaño de una matriz.
`eye`()	Crear matriz Identidad.
`zeros`()	Crear matriz de ceros.
`ones`()	Crear matriz de unos.
`random`	Crear matriz de aleatoria.
`matrix`	Crear tipo de dato `matrix`.
`dot`	Producto punto entre vectores.
`linalg.det`	Determinante de una matriz.
`linalg.inv`	Inversa de una matriz.
`cross()`	Producto cruz entre vectores.
`norm`	Norma de una matriz.
`rank`	Rango de una matriz.
`linalg.matrix_power`	Potencia de una matriz
`linalg.lstsq`	Regresión lineal por mínimos cuadrados.
`linalg.eig`	Función que devuelve los valores propios y los vectores propios de una matriz.

14.10 Conclusiones

A lo largo de este capítulo se cubrieron diversos aspectos del álgebra lineal como son las operaciones para matrices y vectores. Utilizando las bibliotecas con las que cuenta Python. Las funciones y rutinas que se encuentran dentro de la biblioteca `numpy` son de código abierto, lo cual resulta muy atractivo y hace que en la actualidad Python y `numpy` se estén convirtiendo en un referente en el ámbito científico y técnico a nivel mundial.

Existen muchas funciones que no se cubren en este capítulo. Sin embargo, el objetivo es ilustrar que el potencial es muy grande para realizar cálculos numéricos con las bibliotecas que se pueden descargar para Python.

14.11 Ejercicios

1. Obtenga la norma de los siguientes vectores:

   a) $v = (-4, 6, -8)$
   b) $v = (1, 1, 1)$
   c) $v = (-1, -1, -1)$
   d) $v = (\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}})$

2. Encontrar el ángulo que forman los siguientes pares de vectores:

   a) $v = (-4, 6, -8); u = (-12, -5, 0)$
   b) $v = (1, 0, 0); u = (0, 1, 0)$
   c) $v = (-1, -1, -1); u = (3, -2, 16)$

3. Un vector unitario es aquel cuya norma invariablemente es uno. Programe un script que, dado un vector en R^3 dé como resultado su vector unitario asociado.

4. Crear una matriz en Python con tipo de dato `array`:

$$M = \begin{bmatrix} 1 & 4 & 3 & 2 \\ -5 & 6 & 2 & 1 \\ 0 & 0 & 0 & 0 \\ 3 & -2 & 1 & 5 \end{bmatrix}$$

5. Obtener con Python el determinante de las siguientes matrices:

$$A = \begin{bmatrix} 1 & 5 & 2 & 1 \\ -1 & 0 & 1 & 2 \\ 3 & 2 & 4 & -1 \\ 2 & 3 & 6 & 0 \end{bmatrix}$$

$$B = \begin{bmatrix} -2 & 5 & 8 & -4 \\ 4 & -3 & 0 & 0 \\ 0 & 0 & 2 & 2 \\ 0 & 0 & 0 & -1 \end{bmatrix}$$

$$C = \begin{bmatrix} 4 & 0 & 0 & 0 \\ -1 & 1/2 & 0 & 0 \\ 3 & 5 & 3 & 0 \\ -8 & 7 & 0 & -2 \end{bmatrix}$$

$$D = \begin{bmatrix} 4 & -3 & 11 & 1 \\ 5 & 9 & 7 & 2 \\ 2 & 3 & 4 & 1 \\ 5 & 3 & -5 & -9 \end{bmatrix}$$

6. Encontrar la solución del siguiente sistema de ecuaciones lineales:

$$\begin{aligned} 2x_1 - 6x_2 + 12x_3 + 16x_4 &= 70 \\ x_1 - 2x_2 + 6x_3 + 6x_4 &= 64 \\ -1x_1 - 3x_2 + 3x_3 - 7x_4 &= -30 \\ 3x_2 + 4x_3 - 6x_4 &= -26 \end{aligned}$$

7. Balanceo de ecuaciones químicas. Encuentre una ecuación química balanceada para esta reacción Redox entre hidruro de Yodo y ácido sulfúrico.

$$aH_2SO_4 \ + \ bHI \rightarrow cS \ + \ dI_2 \ + \ eH_2O$$

8. Resuelva el siguiente sistema de ecuaciones:

$$\begin{bmatrix} 12 \\ 21 \\ 13 \\ 17 \end{bmatrix} = \begin{bmatrix} 2 & 1 & 9 & 4 \\ 3 & 7 & 8 & 0 \\ 5 & 0 & 7 & 2 \\ 6 & 2 & 8 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$

9. Utilizando la función `lstsq` encuentre la ecuación de una línea recta que se aproxime a los datos de la tabla siguiente:

X	1	3	4	6	8	9	12	14
Y	1	2	4	4	5	7	8	9

10. Una cadena de tiendas departamentales decide llevar a cabo un experimento para medir la influencia sobre las ventas del gasto en publicidad. Se realizaron diferentes variaciones relativas en el gasto en publicidad, se comparó y se observaron las variaciones en los niveles de ventas resultantes en dos periodos semestrales. La tabla adjunta muestra los resultados.

Aumento en el gasto publicitario (%)	3	7	18	14	13	12	10	5
Aumento de las ventas (%)	2.4	7.2	10.3	9.1	10.2	4.1	7.6	3.5

Para un modelo de regresión lineal simple, obtenga la recta de regresión que mejor se ajusta a los datos.